Ultra-low-power, Energy-minimal Computer Architectures

Graham Gobieski

December 2021

PhD Thesis Proposal

Committee: Brandon Lucia, Nathan Beckmann, Tony Nowatzki, Todd Mowry and Kenneth Mai

Computer Science Department
School of Computer Science
Carnegie Mellon University

1 Introduction

Ultra-low-power (ULP) sensor devices are increasingly being deployed for a variety of use-cases in many different environments. The applications are wide-ranging and growing in complexity, including monitoring civil infrastructure, in-body health sensing and tiny, chip-scale satellites. Increasingly these applications rely on sophisticated techniques like on-device machine inference and advanced digital signal processing to reason about sensor data. However, existing systems suffer fundamental inefficiencies that demand solutions across the compute stack: from software that enables sophisticated workloads on ULP devices to new, energy-minimal computer architectures.

Sensing workloads are increasingly sophisticated: Sensor devices collect data from a deployed environment and must process the data to support applications. Processing varies and may entail digital signal processing (DSP), computing statistics, sorting, or sophisticated computations such as machine learning (ML) inference using a deep neural network (DNN). As processing sophistication has increased, sensor device capability also has matured to include high-definition image sensors [1] and multi-sensor arrays [2], increasing sensed data volume.

This shift poses a challenge: how can we perform sophisticated computations on simple, ultra-low-power systems? One design is to offload work by wirelessly transmitting data to a more powerful nearby computer (e.g., at the "edge" or cloud) for processing. In offloading, the more data a sensor produces, the more data the device must communicate. Unfortunately, transmitting data takes much more energy per byte than sensing, storing, or computing on those data [3, 4]. While a high-powered device like a smartphone, with a high-bandwidth, long-range radio, can afford to offload data to the edge or cloud, this is not practical for power-, energy-, and bandwidth-limited sensor devices [3, 5].

Local compute reduces cost of communication: Since offloading is infeasible, the alternative is to process data *locally* on the sensor node itself. My work, Sonic, demonstrates how systems can use commodity off-the-shelf microcontrollers (COTS MCU) to filter sensed data locally so that only meaningful data (as defined by the application) are transmitted. Processing data locally minimizes the high energy cost of communication, reducing energy by $\approx 20\times$ compared to a design that always offloads, but makes the application highly sensitive to the energy-efficiency of computation.

Energy-efficiency is critical to end-to-end system performance: Energy efficiency is the primary determinant of end-to-end system performance in ULP embedded systems. For battery-powered devices [6, 7], energy efficiency determines device lifetime: once a single-charge battery has been depleted the device is dead and it is impractical to replace the battery on millions of deployed devices. Even rechargeable batteries are limited in the number of recharge cycles, and a simple data-logging application can wear out the battery in just a few years [8, 9]. For energy-harvesting devices [10–14], energy efficiency determines device performance. These devices store energy in a capacitor and spend most of their time powered off, waiting for the capacitor to recharge. Greater energy efficiency leads to less time waiting and more time doing useful work [15].

Existing devices are energy-inefficient: However, ULP COTS MCUs used in many deeply embedded sensor nodes (e.g., TI MSP430, ARM M0+ & M4+) are energy-inefficient. These MCUs are general-purpose, programmable devices that support a variety of applications. But this generality comes at at a high power, energy, and performance cost.

Programmability is expensive in two main ways [16–18]. First, *instruction supply* consumes significant energy: in the best case, the energy of an instruction cache hit, and in the worst case, the energy of a main memory read and instruction cache fill. Lacking sophisticated microarchitectural features such as superscalar and out-of-order execution pipelines [19, 20], the energy overhead of instruction supply constitutes a significant fraction of total operating energy. Second, data supply through *register file (RF) access* also consumes significant energy. Together, instruction and data supply can consume 54.4% of the average execution energy across a variety of representative workloads for ULP devices.

Specialization can limit programmability: To combat the energy costs of generality, some recent work has turned to microarchitectural specialization, making a system energy-efficient at the expense of generality and programmability [21–26]. Specialization customizes a system's control and datapath to accommodate a particular workload (e.g., deep neural networks [21, 22]), eliminating inessential inefficiencies like instruction supply and RF access. The downsides of specialization are its high non-recurring engineering cost and its inability to support a wide range of applications. Given the emerging nature of applications in the ULP domain, specialization is premature, so new ULP, energy-efficient, but highly-programmable architectures are needed.

Existing execution models are flawed: Furthermore, both fixed-function ASIC designs and COTS scalar designs assume a trade-off between programmability and energy-efficiency that may be questionable. The fixed-function execution model of ASIC designs limits programmability, while

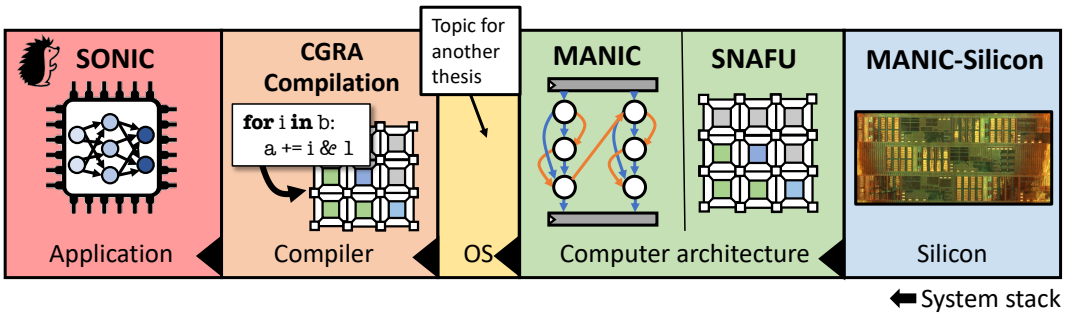


Figure 1: Proposed system stack – from software to silicon – that maximizes energy-efficiency without compromising on programmability and generality.

the scalar execution model of COTS MCUs wastes significant energy. These execution models are at the extremes; there is room in the middle for alternative execution models that balance programmability and energy-efficiency. For example, one starting point is vector execution, which slightly reduces programmability, but amortizes instruction supply energy improving overall energy-efficiency. Developing new execution models, therefore, is critical to resolving the tension between programmability and energy-efficiency.

Objective of this work: The objective of this work is to design a complete system stack that leverages new execution models to maximize energy-efficiency without sacrificing programmability. This approach enables new applications in the ULP domain as improved energy efficiency makes sophisticated workloads practical, while maintaining support for programmability allows for iteration, development of new algorithms, and quick deployment. My existing work addresses two levels of the stack – software and computer architecture – while my proposed work will investigate compilation and silicon implementation. Together these works support the following thesis:

> High energy-efficiency can be achieved across the system stack from software to silicon without compromising on programmability by leveraging new execution models to reduce instruction and data supply energies.

The following contributions form the basis for the thesis and new system stack.

SONIC is the first demonstration of DNN inference on energy-harvesting device: SONIC is an intermittence-aware software system with specialized support for DNN inference. It runs optimized networks found using GENESIS, a tool that automatically compresses networks to balance inference accuracy and energy. SONIC introduces loop continuation, a new technique that dramatically reduces the cost of guaranteeing correct intermittent execution for loop-heavy code like DNN inference. Across three neural networks on a commercially available MCU, SONIC reduces inference energy by 6.9× over the state-of-the-art.

MANIC is energy-efficient vector-dataflow co-processor: MANIC is an efficient vector-dataflow architecture for ultra-low-power embedded systems. It achieves high energy-efficiency without sacrificing programmability and generality. MANIC introduces *vector-dataflow execution*, allowing it to exploit the dataflows in a sequence of vector instructions and amortize instruction fetch and decode

over a whole vector of operations. By forwarding values from producers to consumers, MANIC avoids costly vector register file reads. By carefully scheduling code and avoiding dead register writes, MANIC avoids costly vector register writes. On average, MANIC is 2.8× more energy efficient than a scalar baseline and 38.1% more energy-efficient than a vector baseline.

SNAFU generates ULP CGRAs: SNAFU builds on MANIC's vector-dataflow execution model, generating ULP course-grain reconfigurable arrays (CGRAs) that implement spatial-vector-dataflow execution. In spatial-vector-dataflow execution, a dataflow graph (DFG) is mapped spatially across the fabric of processing elements, applying the same DFG to many input data values, and routing intermediate values directly from producers to consumers. This minimizes instruction and data-movement energy, just like MANIC, and eliminates unnecessary switching activity because operations do not share execution hardware. SNAFU uses 41% less energy and runs 4.4× faster than MANIC.

Proposed work fills out the rest the stack: My proposed work on compilation and silicon implementation round out the new system stack. MANIC-SILICON is a silicon implementation of the MANIC microarchitecture, serving as the ultimate validation of the vector-dataflow execution model. The testchip includes several designs that implement different execution models – from scalar and vector to application-specific. Measuring power and energy directly from the testchip allows for a fair comparison between vector-dataflow and alternative execution models. Further a physical prototype unlocks future research directions in applications and sensor deployments.

At other other end of stack is CGRA compilation. Compiling general-purpose code to a CGRA is a challenging problem. There is a fundamental tension between energy-efficiency, generality, and scalability. This leads to two core research questions. The first has to do with division of responsibility between the compiler/software and the hardware. SNAFU showed that compilation can be scalable with minimal impact on energy-efficiency by making hardware responsible for scheduling in time and the compiler responsible for scheduling in space. But it is unclear how this changes with increased generality. The second research question has to do with programming interface. A functional interface trades generality for energy-efficiency, while an imperative interface (using languages C/C++) may trade energy-efficiency for increased generality. Finding solutions to these two research questions will have major implications for the hardware and the types of programs that can be compiled and run on it.

Outline of remaining sections: The remainder of the proposal is split into four sections. Sec. 2 will describe SONIC, MANIC, and SNAFU, providing the necessary background and context for understanding subsequent sections. Sec. 3 will detail 1) the tape-out and evaluation of the MANIC silicon prototype and 2) the development of a compiler to target SNAFU. Finally, Sec. 4 will provide timelines for proposed work to be completed and Sec. 5 will conclude.

2 Completed Work

The following section provides a summary of completed work, describing several components of the new system stack and establishing necessary context for ongoing and future work. Sec. 2.1 describes deploying deep neural networks (DNNs) on intermittent, energy-harvesting devices and the limitations of existing commodity devices; Sec. 2.2 describes MANIC, a new vector-dataflow architecture to minimize energy; finally, Sec. 2.3 describes SNAFU, a new spatial-vector-dataflow CGRA-generation framework and architecture.

2.1 Deploying DNNs on intermittent embedded devices

Energy-harvesting technology provides a promising platform for future IoT applications. However, since communication is very expensive in these devices, applications will require inference "beyond the edge" to avoid wasting precious energy on pointless communication. Unfortunately, existing ULP, energy-harvesting, sensors are severely resource constrained and suffer frequent power failures. This makes deploying complex applications that rely on machine inference difficult. The following summarizes SONIC, a software runtime system specialized for DNN inference, that overcomes these challenges. SONIC is the first to demonstrate DNN inference on an energy-harvesting system and reduces energy by 6.9× v. the prior-state-the-art. In spite of this, SONIC also exposes the limitations of existing COTS MCUs and execution models, calling for new computer architectures.

Intermittent, energy-harvesting devices enable new applications: Energy-harvesting devices operate using energy extracted (using e.g. solar cells, radio-frequency-harvester, etc) from their environment. This energy can recharge a battery or be buffered in a capacitor. The option to use a capacitor allows these devices to be deployed to environments (e.g. too hot or cold) unsuitable for batteries. Harvested energy is not continuously available, however, so an energy-harvesting device operates intermittently as energy allows. At the same time, many future IoT applications will require frequent decision making, such as when to trigger a battery-draining camera or communicate interesting sensor data. This will require *intelligence beyond the edge* to effectively make use of harvested energy.

The case for local inference: Many applications today offload most computation to the cloud by sending input data to the cloud and waiting for a response. Unfortunately, communication is not free. In fact, on energy-harvesting devices, communication costs orders-of-magnitude more energy than local computation and sensing. These high costs mean that *it is inefficient and impractical for energy-harvesting devices to offload inference to the edge or cloud*, even on today's most efficient network architectures. The solution is SONIC which enables local inference to filter sensor data so only important data is transmitted, minimizing communication.

Why accuracy matters: Inference accuracy determines end-to-end application performance, motivating the use of DNNs. SONIC proposes a high-level analytical model to predict end-to-end performance, dividing energy between sensing, communication, and inference. The model's figure of

merit is the number of interesting sensor readings that can be sent in a fixed amount of harvested energy. This is denoted as IMpJ or interesting messages sent per joule harvested. In the baseline system that offloads all computation, IMpJ is very low because communication dominates the energy budget. Systems with local inference, on the other hand, show large end-to-end benefits on the order of 480×. But, for these gains to be realized in practice, inference must be accurate, and the benefits quickly deteriorate as inference accuracy declines and the systems communicates more false positives.

Commodity MCUs are severely resource constrained: Deep neural networks with high accuracy, however, often require megabytes or gigabytes of memory to store activations and weights. This is in opposition with what a ULP MCU offers – main memory is usually 256KB or less. Thus the first problem to solve is squeezing the NN into device memory without a significant loss of accuracy.

Genesis compresses NNs: Sonic uses Genesis, a neural network architecture search tool, that I developed to compress NNs. Genesis employs three techniques known to reduce memory footprint: 1) it clamps weights close to zero to zero, making weight matrices sparse, 2) it converts 32b floating point to 16b fixed point, cutting memory footprint in half, and 3) it factors weight matrices into several smaller matrices for fully-connected and convolutional layers. By aggressively applying each of these techniques and then fine-tuning the resulting configuration with additional retraining, Genesis reduces an NN's memory footprint of 10s of megabytes to 10s or 100s of kilobytes while minimizing accuracy loss to 5% or less.

Genesis maximizes IMpJ: Of the numerous different feasible configurations of a NN, Genesis selects the configuration that maximizes IMpJ. This choice is non-trivial. True positive, true negative, and inference energy affect end-to-end application performance in ways that are difficult to predict. Simply choosing the most accurate configuration, is insufficient since it may waste too much energy or under-perform other configurations on true positive or true negative rates.

Intermittent operation complicates local inference: Genesis finds the optimal NN for deployment, but intermittent operation is still an obstacle to local inference. An energy-harvesting device operates intermittently slowly accumulating energy in a hardware buffer (e.g., a capacitor) and operating when the buffer is full. The device drains the buffer as it operates, then it turns off and waits for the buffer to fill again. A power failure clears volatile processor state while non-volatile memory persists. Repeated power failures impede progress [27], and may leave memory inconsistent due to partially or repeatedly applied non-volatile memory updates [28]. These progress and consistency issues lead to incorrect behavior that deviates from any continuously-powered execution [29]. Prior work addressed progress and memory consistency using software checkpoints [28, 30, 31], non-volatile processors (NVPs) [32, 33], and programming models based around atomic tasks [34–36]. These systems guarantee correctness but cost performance and energy (as much as 10×). What these systems have missed is the opportunity to *exploit the structure of the computation to lower the cost of guaranteeing correctness.* Sonic relies on the structure of loop-heavy computations like DNN inference to maintain correctness with minimal energy-overhead.

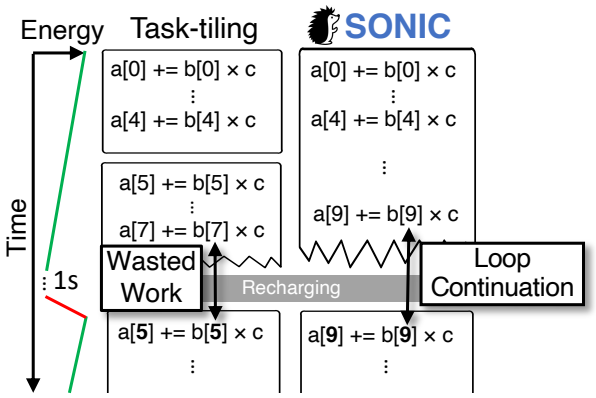


Figure 2: Task-tiling and SONIC's loop continuation mechanism. Loop continuation avoids the re-execution and non-termination costs of task-tiling.

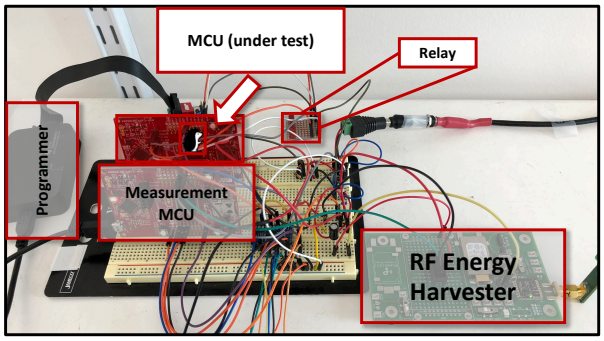
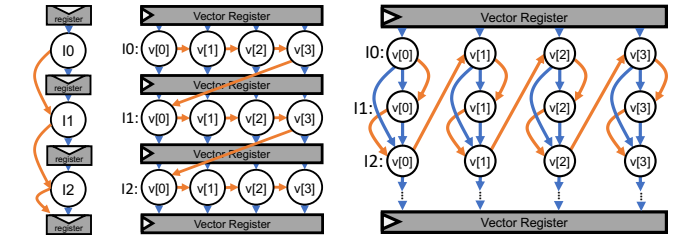


Figure 3: Test setup of SONIC. SONIC runs on the MCU (under test), which operates intermittently from power supplied by the RF-harvester.

SONIC enables inference on intermittent, energy-harvesting devices: SONIC is an intermittence-aware software system with specialized support for DNN inference. SONIC builds on prior work on atomic task-based runtime systems [35] for maintaining correct execution, but avoids specific sources of energy-inefficiency by leveraging computation structure. Specifically, SONIC introduces *loop continuation*, an intermittence-safe optimization that reduces wasted work (work redone after power failure), unnecessary data privatization (arising from undo or redo logging), and atomic task transition overheads (i.e. commits to non-volatile memory). Loop continuation selectively violates the task abstraction for loop index variables, letting SONIC directly modify loop indices without frequent and expensive saving and restoring. By writing loop indices directly to non-volatile memory, SONIC checkpoints its progress after each loop iteration. This is unlike prior work which splits long-running loop iterations into many smaller task-tiles that each execute a fixed number of iterations. Thus, as shown in Fig. 2, following a power interruption, loop continuation allows execution to resume where it left off – the ninth loop iteration – rather than restarting every n-th (fifth in this case) iteration like task-tiling does. Further, loop continuation is safe because SONIC ensures that each loop iteration is idempotent, enforcing that the same data is never read and written to in the same iteration.

SONIC is efficient on commodity hardware: I evaluated SONIC running three neural networks on an MSP430 development board powered by capacitor storing energy harvested by a Powercast radio-frequency harvester (shown in Fig. 3). I compared SONIC to a state-of-the-art task-based runtime system that splits work into tiles. SONIC is the first demonstration of an NN running on an intermittent, energy-harvesting device and uses 6.9× less energy than the task-tiling system.

Commodity hardware is fundamentally energy-inefficient: The experience building SONIC exposed the limitations of existing commodity microcontrollers. Some of these limitations are superficial – the MSP430 has no caches, no on-chip multiplier and does not implement vector left shift. However, these devices are also fundamentally flawed in their execution model. They implement



(a) Scalar *(b) Vector* *(c) Vector-dataflow*

Figure 4: Different execution models. Orange arrows represent control flow, blue arrows represent dataflow. MANIC relies on vector-dataflow execution, avoiding register accesses by forwarding and renaming.

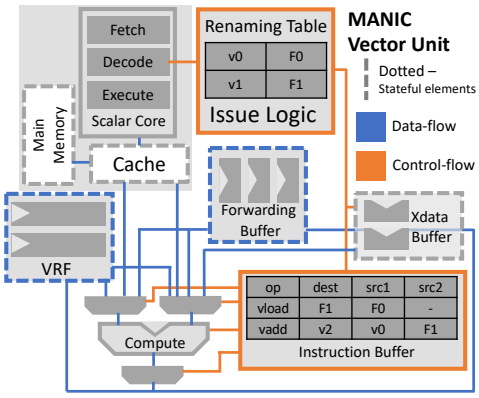


Figure 5: A block diagram of MANIC's microarchitecture (non-gray region).

scalar execution which burns significant amount of energy fetching and decoding instructions ($\approx 40\%$ of total energy) as well as communicating intermediates between instructions through the register file ($\approx 20\%$ of total energy). Clearly ASIC designs would reduce or eliminate these sources of inefficiency, but would also compromise on flexibility and programmability. Choosing the correct execution model therefore is critical to balancing flexibility/programmability with energy-efficiency.

2.2 MANIC: an ULP, vector-dataflow co-processor

MANIC is the solution to the problems identified with existing commodity microcontrollers. MANIC is an ULP, vector-dataflow co-processor that combines vector and dataflow execution to reduce instruction supply and vector register file (VRF) energies. The following describes the vector-dataflow execution model, how MANIC reduces VRF accesses, the microarchitecture of MANIC and compares MANIC to existing designs.

Vector-dataflow execution reduces energy by combining vector and dataflow execution: MANIC introduces vector-dataflow execution. Vector-dataflow execution provides general-purpose programmability, while minimizing instruction and data supply overheads. It combines vector execution with dataflow instruction fusion. In vector execution, vector instructions specify an operation that applies to an entire vector of input operands (as in ample prior work). The key advantage of this is that control overheads imposed by each instruction – instruction cache access, fetch, decode, and issue – amortize over the many operands in the vector of inputs. Fig. 4 illustrates the difference between scalar execution and vector execution. Blue arrows show dataflow and orange arrows show control flow. Fig. 4a shows scalar execution, which repeatedly fetches and decodes the same instructions. Fig. 4b shows vector execution. The execution fetches three vector instructions, which each perform the same operation across the sequence of vector operands. This amortizes the control overhead of a scalar execution, reducing instruction supply energy.

The other aspect to vector-dataflow execution is dataflow instruction fusion. Dataflow instruction fusion identifies windows of contiguous, dependent vector instructions. Dataflow instruction fusion eliminates register file reads by directly forwarding values between instructions within the window.

This reduces data supply energy compared to vector execution which performs two vector register file reads per instruction. Accessing the vector register file has an extremely high energy cost that scales poorly with the number of access ports [18, 37]. Fig. 4c shows vector-dataflow execution. Unlike vector execution, control proceeds *vertically* then *horizonitally*. In the figure, MANIC executes the first element (`v[0]`) across instructions `I0`, `I1`, and `I2`. This enables dataflow forwarding from `I0` to (shown with blue arrows) to `I2` without accessing the VRF. Then control steps horizontally, executing the same window of operations on the next element of the vector, `v[1]`. In this way, dataflow instruction fusion eliminates VRF reads and reduces data supply energy.

Kill annotations reduce VRF writes: MANIC also eliminates a majority of VRF writes using compiler annotations to hint to the hardware when a VRF write is unnecessary. The compiler identifies when a value is *dead* and will no longer be used. In these cases, it marks the value using a kill annotation, telling the hardware to skip the VRF write. Since values are usually finally consumed shortly after being produced, kill annotations eliminate most VRF writes and significantly reduce VRF energy.

Microarchitecture of MANIC: Fig. 5 shows the design of MANIC. MANIC adds four components to a simple vector core to support vector-dataflow execution: (1) issue logic and a register renaming table; (2) an instruction window buffer; (3) an xdata buffer and (4) a forwarding buffer.

MANIC's issue logic is primarily responsible for creating a window of instructions to execute according to vector-dataflow. The issue logic activates once per window of instructions, identifying, preparing, and issuing for execution a window of dependent instructions over an entire vector of inputs. The issue logic identifies dataflow between instructions by comparing the names of their input and output operands. If two instructions are dependent – the output of one of the instructions is the input of another – MANIC renames the instructions' register operands to refer to a free location in MANIC's forwarding buffer, instead of to the register file. The issue logic records the renaming in MANIC's renaming table, a small, fixed-size, directly-indexed table. After identifying dependent operations in a window, the issue logic dispatches the window of operations for execution.

MANIC buffers instructions with renamed operands in the instruction window. The instruction window determines what operation MANIC's single functional unit should execute next. A key feature of the instruction window's control logic is its ability to select an operand's source or destination. For input operands, the instruction window controls whether to fetch an operand from the VRF or from MANIC's forwarding buffer. Likewise, for output operands, the instruction window controls whether to write an output operand to the VRF, to the forwarding buffer, or to both.

In addition to the information in the instruction buffer, some operations like vector loads and stores require extra data (e.g. base address and stride) available from the scalar register file when the instruction is decoded. Due to the loosely coupled nature of MANIC, this extra data must be buffered alongside the vector instruction. Since not all vector instructions require values from the scalar register file, MANIC includes a separate buffer, called the xdata buffer, to hold this extra data.

Finally, the forwarding buffer is a small, directly-indexed buffer that stores intermediate values

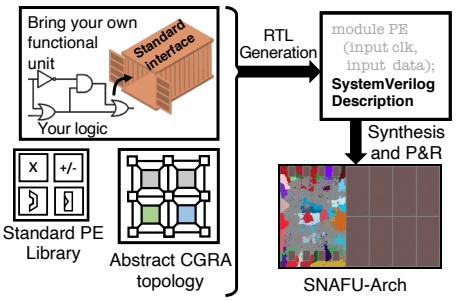


Figure 6: Overview of SNAFU. SNAFU is a flexible framework for generating ULP CGRAs.

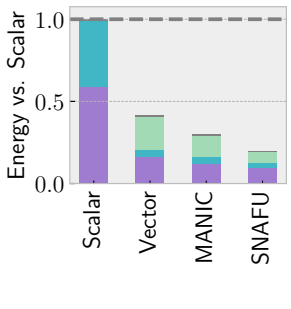


Figure 7: Average energy of scalar, vector, MANIC, and SNAFU.

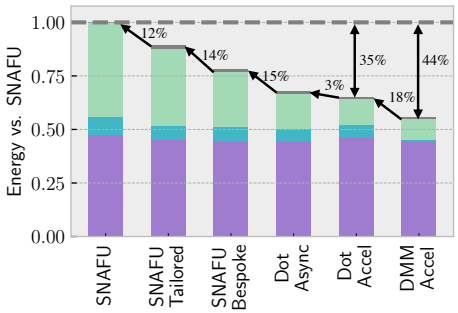


Figure 8: Quantifying the cost of SNAFU's programmability on DMM benchmark.

as MANIC's execution unit forwards them to dependent instructions in the instruction window. The issue logic lazily allocates space in the forwarding buffer and renames instruction's forwarded operands to refer to these allocated entries. The benefit of the forwarding buffer is that it is very small and simple, which corresponds to a very low static power and access energy compared to the very high static power and access energy of the vector register file. By accessing the forwarding buffer instead of accessing the VRF, an instruction with one or more forwarded operands consumes less energy than one that executes without MANIC.

MANIC is energy-efficient: I evaluated MANIC in a complete system (including SRAM main memory and RISC-V scalar core) built entirely in RTL including SRAM macros and using industry-grade CAD tools to estimate power via a post-synthesis annotated switching model. I compared MANIC to a scalar core baseline as well as a vector baseline, which computes vector instruction-by-instruction, reading and writing vector outputs to a VRF. MANIC uses 2.8× and 38.1% less energy while achieving similar performance than the scalar and vector designs, respectively. To further validate the MANIC architecture, MANIC is being taped-out so that energy can be measured directly from a silicon prototype. A broader discussion of the tape-out can be found in Sec. 3.1.

2.3 SNAFU: an energy-minimal CGRA-generation framework and architecture

SNAFU is an ULP CGRA generation framework and architecture that improves upon MANIC by implementing spatial-vector-dataflow execution. The following describes SNAFU in detail – starting with the SNAFU framework, then describing how SNAFU minimizes energy and finally, how SNAFU compares to MANIC.

SNAFU generates ULP CGRAs: SNAFU is a framework for generating energy-minimal, ULP course grain reconfigurable arrays and compiling applications to run efficiently on them. The generated CGRA is comprised of a set of heterogeneous processing elements connected to each other via an on-chip network. The architecture is coarse in that the PEs support higher-level operations, like multiplication, on multi-bit data words, as opposed to bit-level configurability in FPGAs. The PEs can be configured by the compiler to perform different operations and the NoC can be configured

to route values directly between PEs.

Fig. 6 shows SNAFU's workflow. SNAFU takes two inputs: a library of processing elements (PEs) and a high-level description of the CGRA topology. SNAFU lets designers customize the ULP CGRA via a *"bring your own functional unit"* approach, defining a generic PE interface that makes it easy to add custom logic to a generated CGRA.

With these inputs, SNAFU generates complete RTL for the CGRA. This RTL includes a statically routed, bufferless, multi-hop on-chip network parameterized by the topology description. It also includes hardware to handle variable-latency timing and asynchronous dataflow firing. Finally, SNAFU simplifies hardware generation by supporting top-down synthesis, making it easy to go from a high-level CGRA description to a placed-and-routed ULP design ready for tape out.

SNAFU minimizes energy from the ground-up: SNAFU is designed from the ground-up to maximize energy-efficiency in three ways: 1) by implementing spatial-vector-dataflow, 2) by supporting asynchronous dataflow firing without tag-token matching, and 3) by minimizing buffers in the on-chip network.

SNAFU reduces energy by implementing *spatial* vector-dataflow execution. Like vector-dataflow, SNAFU's CGRA amortizes a single fabric configuration across many computations (vector), and routes intermediate values directly between operations (dataflow). But SNAFU *spatially* implements vector-dataflow: SNAFU *buffers intermediate values locally* in each PE (vs. MANIC's shared forwarding buffer) and *each PE performs a single operation* (vs. MANIC's shared pipeline). This minimizes switching activity in shared pipeline resources, a significant sink of energy in MANIC. In fact, the reduction in switching activity in SNAFU accounts for the majority of the 41% of energy savings that SNAFU achieves vs. MANIC.

SNAFU also supports asynchronous dataflow firing without expensive tag-token matching. Prior CGRAs have explored both static and dynamic strategies to assign operations to PEs and to schedule operations [38]. Static assignment and scheduling is most energy-efficient, whereas fully dynamic designs require expensive tag-matching hardware to associate operands with their operation. SNAFU is designed to easily integrate new PEs with unknown or variable latency, so a fully static design is not well-suited to SNAFU, but SNAFU cannot afford full tag-token matching either. SNAFU's solution is a hybrid CGRA with static PE assignment and dynamic scheduling. Each PE uses local, asynchronous dataflow firing to tolerate variable latency, avoiding tag-matching by enforcing that values arrive in-order. This design lets SNAFU integrate arbitrary PEs with little energy or area overhead ($< 2\%$ system energy).

Lastly, SNAFU includes a statically-configured, bufferless, multi-hop on-chip network designed for high routability at minimal energy. Static circuit-switching eliminates expensive lookup tables and flow-control mechanisms without degrading performance (as showed by [39]). The network is also bufferless (a PE buffers values it produces), eliminating the NoC's primary energy sink (half of NoC energy or more [40]).

SNAFU is faster and more energy-efficient than MANIC: I used SNAFU to generate SNAFU-

ARCH, a 6x6 mesh with four multiply units, eight scratchpad units, twelve memory units, and twelve basic ALU units. I synthesized SNAFU using an industrial sub-28nm, high-threshold-voltage FinFet PDK including compiled memories. I compared SNAFU to MANIC, to a scalar baseline, and to a vector baseline across ten different benchmarks using a post-synthesis annotated switching model to estimate power. Fig. 7 shows the average normalized energy (normalized to scalar baseline) of the different designs across the benchmarks. On average, SNAFU uses 81%, 57%, and 41% less energy and is 9.9×, 3.2×, and 4.4× faster than the scalar, vector, and MANIC designs, respectively.

SNAFU's flexibility comes at minimal cost: To compare SNAFU to an ASIC, I conducted three case-studies on FFT, Sort, and DMM that compare SNAFU to fully-custom ASIC designs in the same technology node. Additionally, I implemented four intermediate designs between SNAFU and the ASIC designs to determine sources of inefficiency in SNAFU. Fig. 8 shows the case-study for DMM. From left-to-right are: SNAFU, SNAFU-Tailored which customizes the fabric for DMM, SNAFU-Bespoke which removes reconfigurability, Dot-Async which is a dot product accelerator with asynchronous dataflow firing, Dot-Accel, a dot-product accelerator, and finally DMM, a dense matrix-matrix multiply accelerator. SNAFU is within 2.6× of the energy of the ASIC design without sacrificing programmability. Looking deeper, these results show that software programmability (SNAFU-Tailored uses 14% more energy than SNAFU-Bespoke) and asynchronous dataflow firing (Dot-Accel uses 3% more than Dot) costs minimal energy and that there is opportunity to improve overall system efficiency by accelerating larger portions of computation with SNAFU (SNAFU at the moment only accelerates inner-loops like dot-product).

3 Ongoing and Future work

My prior work is unified on the pursuit of energy-efficiency without sacrificing programmability and with a focus on software and computer architecture. My ongoing and proposed work will follow in a similar vein, rounding out a new energy-efficient system stack. At the low-level, I will tape-out MANIC in order to validate the vector-dataflow execution model by measuring energy directly from a physical prototype. At the other end of the stack, I will investigate compiler techniques to target SNAFU-like architectures, improving overall system efficiency by further offloading computation from a scalar core to a SNAFU-like CGRA.

3.1 MANIC-SILICON

MANIC-SILICON is a complete silicon prototype that includes the MANIC co-processor discussed previously. It has been under development for nearly two years – taping out a custom chip is a significant undertaking. Recently I received testchips back from Intel and characterization of the silicon is ongoing. The following paragraphs discuss the objectives of the testchip, the design, verification and bring-up of the testchip and concludes with a summary of measured results.

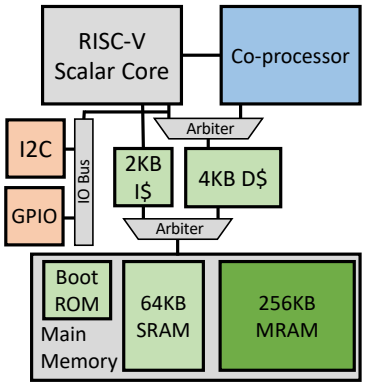


Figure 9: Block diagram.

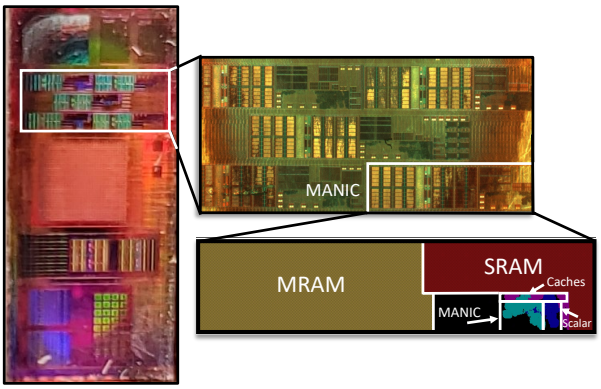


Figure 10: Testchip electron micrograph.

3.1.1 Objectives and design of MANIC-SILICON

MANIC-SILICON has two objectives: 1) be a viable replacement for existing MCUs in ULP sensor deployments and 2) validate vector-dataflow execution against competing execution models. Towards the first objective, MANIC-SILICON meets the criteria for remote ULP sensor deployments; specifically, the testchip operates at extreme low-power, supports general-purpose programs, can run standalone (without an external FPGA or MCU driving control), has I2C and GPIO to communicate with sensors and integrates a non-volatile main memory, a requirement for devices that may suffer (frequent) power failures. Towards the second objective, the MANIC-SILICON testchip includes several independent designs that implement different execution models for comparison. They include scalar, vector, vector-dataflow, and application-specific. The following paragraph describes each design in more detail.

MANIC-SILICON validates vector-dataflow against competing execution models: MANIC-SILICON includes five different designs: a scalar design, an optimized vector design, MANIC, the vector design with a custom VRF (custom SRAM macro), and a design that includes three different accelerators. Each design follows the block diagram in Fig. 9, possessing a RISC-V (RV32emi) microcontroller, a 2-KB instruction cache, a 4-KB data cache, a module that handles I2C communication and programming, a module that handles GPIO communication, a module that tracks statistics about device operation, and main memory composed of 64KB of SRAM, 1KB of ROM, and 256KB of embedded MRAM (eMRAM). The scalar design adds no additional components to this common core. The vector design adds a simple vector co-processor. The co-processor has a three-stage pipeline (VIssue, VExecute, VWriteback) that computes a vector operation by iterating over vector elements. The MANIC design adds the MANIC co-processor. The vector design with the custom VRF replaces the VRF made from standard Intel SRAM macros in the vector design with custom SRAM macros; instead of using two banks of 1r1w Intel SRAM macros to emulate a 2r1w macro, the custom VRF is made up of true 2r1w custom SRAM macros. Lastly, the accelerator design has three accelerator: a sparse-matrix-dense-vector multiply accelerator, a sort accelerator, and a dense convolution accelerator. Each design serves as comparison point: the scalar design is similar to existing commodity devices, the vector designs implement vector execution, providing the closest

competition to MANIC, and the accelerator design is representative of ASIC designs, providing the upper-bound in energy-efficiency.

Differences in implementation of MANIC: MANIC-SILICON's implementation of the MANIC microarchitecture slightly differs from what was previously discussed in Sec. 2.2 and that which is reported in [41]. Low-level simulations of entire designs showed that MANIC's shared pipeline resources toggled more often and burned more power than the vector baseline's. This is a direct result of the implementation of the vector-dataflow execution model where different operations time multiplex on the same resources. To reduce toggling, the basic three-stage pipeline (VIssue, VExecute, VWriteback) is augmented with two additional stages – VGate and VMemory. VGate comes before VExecute, while VMemory is before VWriteback. VGate determines the source for each operand – i.e., from the VRF, the forwarding RF, or pipeline forwarding paths – and steers operands to the multiplier or ALU in the VExecute stage. This prevents unnecessary toggling on the multiplier or ALU. VMemory in a similar way prevents toggling on memory ports.

Why eMRAM?: Each of the designs on MANIC-SILICON include 256KB of embedded MRAM. This is a requirement for remote deployments where energy is sparse and a device may suffer frequent power failures. MANIC-SILICON is also one of the first demonstrations of eMRAM integrated into a complete system. eMRAM provides non-volatility at lower costs than competing NVM technologies. Compared to flash, eMRAM has word-level addressibility, higher write endurance, lower read and write latencies, and lower read and write energies. Also since it can be fabricated in the same process as logic the use of eMRAM avoids expensive off-chip IO.

3.1.2 Verification and bring-up of MANIC-SILICON

Verification of chip design is the most important step in the tape-out process and showed its value with MANIC-SILICON as bring-up took just three days. Verification for MANIC-SILICON involved three items: 1) integration of design-for-test (DFT) structures to allow for easy debugging, 2) safeguards in the event certain features do not work and 3) comprehensive unit and integration testing at each level of hardware abstraction (e.g pre- and post- synthesis and post-place-and-route). The following paragraphs describe the design-for-test strategy and chip bring-up in more detail.

Design-for-test: DFT is a design methodology that makes a design robust to feature failure. For MANIC-SILICON, I developed a series of standard DFT modules with scan-chain interfaces that wrap selectively-chosen registers, SRAM macros, and the eMRAM macro. This allows each of these structures to be read and written to directly while interactively minimally with on-chip logic. There are two important examples of note in MANIC-SILICON. First, entire features can be disabled and signals routed around them; e.g. the instruction and data caches can both be disabled. Second, there are five mechanisms for programming the chip. The first two mechanisms rely on the ROM-based bootloader. If eMRAM is enabled program code and state is stored in the eMRAM otherwise its stored in the SRAM. The second two mechanisms use the scan-chain interfaces to the SRAM

and eMRAM macros. Program code and state can be written directly to these macros and the bootloader directed to start execution immediately (without flashing) from either the SRAM or eMRAM. Finally, in the worst case scenario there is also a mechanism for feeding the RISC-V core directly with instructions and data for loads. These built-in safeguards are not only important for debugging, but also protect chip operation from a number of problematic scenarios when features might have failed.

Programming and communication with the chip: To program and communicate with the Manic-Silicon prototype, I co-developed an Arduino-based programmer with Manic-Silicon's bootloader using an FPGA implementation of Manic-Silicon. The Arduino-based programmer converts serial commands to I2C commands; passing data to and from the computer to the testchip. Programming is initiated by the testchip which asks the programmer for the application size. The programmer then communicates with the computer to get the application's binary size and responds to the testchip over I2C. Then the testchip repeatedly asks the programmer for bytes of the application binary. The programmer receives this data from the computer and responds to the testchip. There is a handshaking protocol between both the programmer and the computer as well as the programmer and the testchip. This ensures data is not lost during flashing/programming. Once all data has been transferred, the testchip jumps to the starting address in main memory. The programmer continues to be connected, handling further communication (primarily printing to console) between the testchip and the computer.

Tuning the eMRAM: Besides programming, the other important item for chip bring-up is the tuning of the embedded MRAM macro. Due to manufacturing variability, configuration of the eMRAM is different chip-to-chip. As such there are a number of different settings and parameters to get the eMRAM macro reading and writing correct data as well as minimizing write latency. A collaborator helped build a tool to quickly sweep the configuration space (using the scan-chain interface with the eMRAM) to determine the optimal settings for a particular instance of the macro.

3.1.3 Characterization of silicon

Manic-Silicon achieves state-of-the-art energy-efficiency and ultra-low-power operation. The following paragraphs describe the measurement setup and characterization of the silicon in detail.

Measuring power from the chip: Manic-Silicon is designed with measuring power in mind. Logic, SRAM, and eMRAM are on separate supply rails so that their contribution to total power can be quantified. Further, the five design in Manic are isolated from each other in the Logic and SRAM domains. In other words, power can be supplied to the logic and SRAM of a single design at a time.

To measure power, I measure current using a digital voltmeter in series and/or a source meter unit. I sweep clock frequency and SRAM and logic voltages to determine the energy-minimal configuration.

Manic-Silicon achieves 256MOPS/mW using $19\mu W$ ***@ 4MHz:*** While further characterization is ongoing, Manic-Silicon achieves state-of-the-art efficiency and validates vector-dataflow execution and the Manic architecture. Fig. 10 shows an electron micrograph of Manic-Silicon testchip. With MRAM disabled, Manic-Silicon achieves a peak efficiency of 256MOPS/mW at ultra-low-power consumption of $19\mu W$ at 4MHz. This is 2.6× higher than the prior state-of-the-art low-power optimized microcontroller. Further compared to the vector design, vector-dataflow execution saves 12% energy (with caches disabled) on average across ten benchmarks by eliminating frequent accesses to the VRF.

Manic-Silicon is also one of the first demonstrations of eMRAM integrated into a complete design. With eMRAM enabled, leakage and dynamic power of the macro dominate overall system power. Leakage of SRAM and the logic together is on the order of several microwatts, but leakage of MRAM is $0.7 - 1.1mW$ (range due to chip manufacturing variability and data stored in MRAM). Further, dynamic power can be another $1 - 1.5mW$. The caches are important to reduce dynamic power and energy by minimizing accesses to the eMRAM, but limit clock frequency, which puts an upper bound on efficiency given the high leakage of the macro. With that said, a system with eMRAM enabled still achieves $11MOPS/mW$.

Ongoing characterization and future opportunities: Characterization of Manic-Silicon is ongoing, focusing on comprehensively evaluating the Manic design against baselines across different benchmarks and applications. See the next section for a detailed timeline of the development and characterization of Manic-Silicon. Additionally, Manic-Silicon offers two future research directions to explore: 1) deployment of the hardware to a remote environment and 2) fine-grain (power-)gating of MRAM and memory management. The first direction is under investigation by a colleague, who is developing a custom PCB for Manic-Silicon so that Manic-Silicon can be launched into space as part of a chipsat. The second direction would improve the energy-efficiency of Manic-Silicon when eMRAM is enabled. eMRAM uses significantly more leakage and dynamic power than logic and SRAM, so minimizing the amount of time and accesses to it is fundamental to improving energy-efficiency. Fortunately, Manic has 64KB of SRAM that can be used as a scratchpad. Either applications need to be rewritten to effectively use the SRAM or a compiler pass needs to be developed to automatically manage memory between the volatile and non-volatile segments. The energy and latency costs of turning on and off the MRAM will dictate the rate at which the MRAM can be gated and the granularity of the data stored in SRAM.

3.2 CGRA Compilation

Compiling code to target a CGRA is a challenging research problem, but is important to address since it can improve programmer productivity and overall system efficiency. The problem is framed by three questions: 1) what is the programming abstraction and how does it affect compilation and efficiency, 2) how to scale the compiler (and fabric) to handle programs with hundreds or thousands of operations, and 3) what are the required constraints on a program for efficient execution on a reconfigurable fabric.

The following paragraphs motivate the need for a compiler and then discuss each question in detail.

Compilation improves overall system efficiency by increasing programmer productivity: A compiler improves programmer productivity by raising the programming abstraction. But for SNAFU-like CGRAs it also will improve overall system energy-efficiency. This is because SNAFU-like CGRAs use 81% less energy than scalar designs, meaning that if more work is pushed to the CGRA fabric from the scalar core, overall system efficiency increases. By raising the abstraction from low-level vector assembly operations to higher-level code constructs, the efficiency of SNAFU-like CGRAs is more accessible to the programmer.

More than auto-vectorization: However, compilation for SNAFU-like CGRAs is more than just auto-vectorization. While SNAFU's compiler only supports low-level vector operations perfect for vectorizable inner-loops without complex control-flow, there is opportunity to accelerate outer-loops to improve efficiency even further. The case study in Fig. 8 provides evidence: the dot-product accelerator uses 18% more energy than the dense-matrix-matrix multiply accelerator, which accelerates outer-loops. This suggests that further efficiency can be unlocked by compiling outer-loops and more complex control-flow constructs to the CGRA fabric.

I now turn my attention to the three previously-mentioned research questions that frame the development of a compiler to target SNAFU-like CGRAs.

What is the programming abstraction?: The first question to explore has to do with the programming abstraction because it affects the latter two questions as well as energy-efficiency. SNAFU's existing compiler exposes a low-level vector assembly interface, which is sufficient for programs that are easily vectorized. But not all programs can be easily transformed to vector operations and they may require expert-level knowledge to do so. Instead a different abstraction can be provided. There are two general approaches: one based on imperative languages and the other based on a functional approach utilizing parallel patterns.

Wavescalar [42], TRIPS [43], and SGMF [44] provide an imperative interface, relying on compiler passes to extract dataflow graphs that can be scheduled onto hardware. They co-design the reconfigurable fabric hardware with the compiler to improve efficiency and natively support constraints on program order that the imperative interface assumes (e.g. load-store ordering). However, there are two challenges with this approach. First, dataflow graphs extracted directly from imperative programs tend to be larger and more complex than programs/graphs using parallel patterns. This makes the mapping of dataflow graphs nodes (operations) to PEs more difficult. Second, the imperative interface makes memory alias disambiguation difficult or in many cases impossible. This can restrict parallelism and by extension performance because the compiler is overly conservative since in many cases it cannot determine whether loop iterations are independent or not. It can also mean that extra hardware is required to check that operations execute in order with the correct operands. The hardware buffers intermediates in queues, tags data, and checks the tags of inputs to make sure they match before enabling an operation. This comes at the expense of energy.

The alternative approach uses parallel patterns. Parallel patterns like map, fold, reduce, etc.

capture common loop idioms. They also make explicit loop-carry dependencies and the ordering (or lack thereof) of loop iterations. This makes it possible to decompose a parallel pattern into many smaller independent tasks, which can effectively utilize hardware resources. Plasticine [45] and Softbrain [46] rely on this property to simplify compilation and maximize performance. However, there are limitations to parallel patterns. Applications written using imperative languages need to be translated to using parallel patterns. This can sometimes be difficult or impossible depending on the patterns supported and application logic. For example, loops with a bound that changes during iteration (e.g. outer-loop in naive BFS) are challenging; this relatively common pattern makes the static division of work nearly impossible. Moreover, there is no exhaustive list of parallel patterns, which means that applications might rely on patterns not (efficiently) supported by hardware. Follow-up work [47, 48] to Plasticine had to add support for different patterns in order to accelerate database and sparse workloads.

Is CGRA compilation even practical?: The next framing question has to do with the scalability of compiler as application logic complexity increases. It is deeply dependent on the division of responsibility between software and hardware. Prior work has struggled to find the balance. Some work on systolic arrays places responsibility wholly on software to produce static schedules for space and time. These architectures require that the timing of all operations be known before execution so caches are generally not supported. Also, they rely on complicated compilers that do not scale as they have to unroll loops (sometimes without limit) to discover initiation intervals for high utilization of hardware resources.

Other prior work puts all responsibility on the hardware. These are mainly out-of-order core designs that dynamically schedule operations, rename registers and allocate functional units on-the-fly. This requires significant hardware (e.g. reorder buffer, store-queue, etc.), which limits energy-efficiency.

Finally, some prior work divides the scheduling problem. This is the secret to Snafu's compiler scalability – the software schedules in space, while the hardware schedules in time. Specifically, supporting asynchronous dataflow adds minimal energy overhead, but greatly simplifies the compilation problem. Maintaining this balance is therefore critical to future Snafu-like CGRAs that accelerate more than just vectorizable, inner-loop nests.

What are the constraints on a DFG to maximize energy-efficiency?: The last framing question asks what constraints on a program's dataflow graph are needed to maximize energy-efficiency while preserving correctness and without sacrificing performance. The implementation of conditional execution is a motivating example for this question. Conditional execution can be implemented in three ways: phi, inverse-phi, and predication. Each adds a different set of dependencies to a program's dataflow graph.

Phi gates take two data inputs and a selector, that chooses between the data inputs, producing a single data output. These gates maximize performance at the cost of energy-efficiency; both sides of a branch produce results with the phi gate discarding one of them.

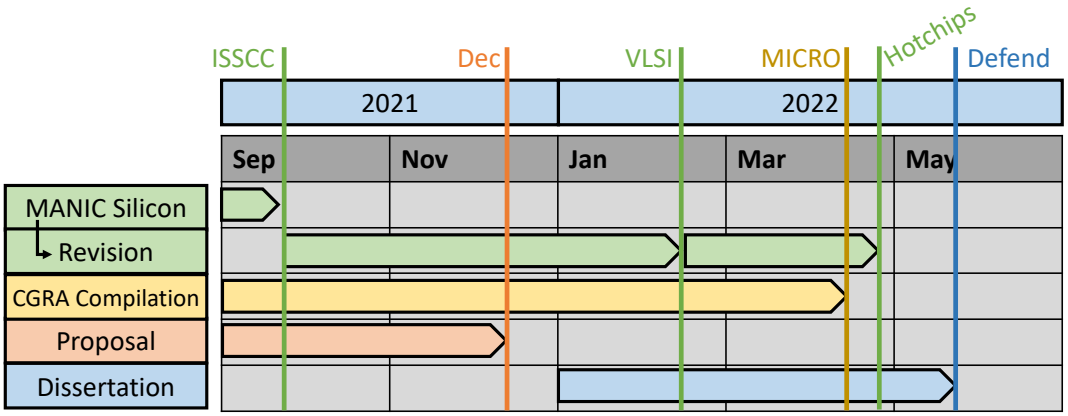


Figure 11: Timeline of work.

Inverse-phi are the complement to phi gates. They take a single data input and a selector; the gate has two outputs with the selector choosing between the outputs for sending input data. This approach minimizes energy by disabling one side of the branch, but does not achieve maximal performance since the branch condition cannot be evaluated in parallel.

Finally, there is predication. Predication adds a dependence between the sides of a branch, forwarding data from one side of the branch to other. If the condition is true then the true branch fires and the false branch passes through the results of the true branch; if the condition is false the true branch still fires but sends dummy data that the false branch ignores, sending the results of its operations instead. Predication turns branching code into straightline code, balancing energy-efficiency with performance.

Thus, implementation matters. Phi gates, inverse-phi gates, and predication place different constraints on a program's dataflow graph, which has implications for performance and energy-efficiency.

Ongoing work: Compiling programs for SNAFU-like CGRAs is in its exploratory phase. The three framing research questions are guiding development. Along with a colleague, I am investigating using LLVM to extract and transform a program's dataflow graph, which can be scheduled on CGRA hardware. Please see the following section for a timeline.

4 Timeline

Fig. 11 shows a detailed timeline for proposed work and my thesis dissertation and defense. I submitted MANIC-SILICON in September and will conduct further characterization of the testchip in preparation for resubmission to VLSI in February (and/or HotChips in April). The work on CGRA compilation is less developed. I am working with a colleague to push forward development and a submission to MICRO in late March might be possible. Finally, I plan on starting work on my dissertation at the new year and am aiming to defend in the Spring.

5 Conclusion

My work contributes to a new system stack that maximizes energy-efficiency without sacrificing programmability from software to silicon. SONIC is a software runtime system that enables machine inference on intermittent, energy-harvesting devices. MANIC is a vector-dataflow co-processor that is far more efficient than existing COTS MCUs. SNAFU builds on MANIC, generating ULP CGRAs that are even more efficient. MANIC-SILICON is a silicon prototype of the MANIC architecture that validates vector-dataflow execution. And a compiler to target SNAFU-like hardware will increase programmer productivity and overall system energy-efficiency. Taken together these projects will enable new applications in the ULP domain.

References

[1] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith, "Towards battery-free {HD} video streaming," in *NSDI 15*, 2018.

[2] G. Laput, Y. Zhang, and C. Harrison, "Synthetic sensors: Towards general-purpose sensing," in *Proc. of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017.

[3] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *ASPLOS*, 2019.

[4] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi, "Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet," in *MobiSys 2*, (New York, NY, USA), ACM, 2004.

[5] A. Dongare, C. Hesling, K. Bhatia, A. Balanuta, R. L. Pereira, B. Iannucci, and A. Rowe, "Openchirp: A low-power wide-area networking architecture," in *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*, 2017.

[6] D. Culler, J. Hill, M. Horton, K. Pister, R. Szewczyk, and A. Wood, "Mica: The commercialization of microsensor motes," *Sensor Technology and Design, April*, 2002.

[7] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. Moura, and L. Soibelman, "Sensor andrew: Large-scale campus-wide sensing and actuation," *IBM Journal of Research and Development*, vol. 55, no. 1.2, 2011.

[8] N. Jackson, "lab11/permamote," Apr 2019.

[9] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *ENSSys 7*, 2019.

[10] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *ASPLOS 23*, 2018.

[11] J. Hester, L. Sitanayah, and J. Sorber, "Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors," in *Proc. of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015.

[12] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet of things," in *SenSys 15*, 2017.

[13] H. Zhang, J. Gummeson, B. Ransford, and K. Fu, "Moo: A batteryless computational rfid and sensing platform," *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep*, 2011.

[14] A. Wickramasinghe, D. Ranasinghe, and A. Sample, "Windware: Supporting ubiquitous computing with passive sensor enabled rfid," in *RFID*, April 2014.

[15] H. Desai and B. Lucia, "A power-aware heterogeneous architecture scaling model for energy-harvesting computers," *IEEE Computer Architecture Letters*, vol. 19, no. 1, 2020.

[16] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[17] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38, 2010.

[18] J. D. Balfour, W. J. Dally, M. Horowitz, and C. Kozyrakis, *Efficient embedded computing*. PhD thesis, 2010.

[19] T. Instruments, "Msp430fr5994 sla," 2017.

[20] A. Traber, "Pulpino: A small single-core risc-v soc,"

[21] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,"

[22] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[23] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor,"

[24] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, vol. 43, 2015.

[25] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[26] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, 2010.

[27] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *ASPLOS*, Mar. 2011.

[28] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), ACM, 2015.

[29] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *SIGOPS Oper. Syst. Rev.*, vol. 50, Mar. 2016.

[30] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proc. of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[31] M. Hicks, "Clank: Architectural support for intermittent computation," in *Proc. of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), ACM, 2017.

[32] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[33] K. Ma, X. Li, J. Li, Y. Liu, Y. Xie, J. Sampson, M. T. Kandemir, and V. Narayanan, "Incidental computing on iot nonvolatile processors," in *Proc. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[34] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2016.

[35] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, (Vancouver, BC, Canada), ACM, Oct. 22–27, 2017.

[36] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermi!ently powered ba!eryless sensors," in *Proc. of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17.

[37] C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, 2003.

[38] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2020.

[39] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC*, 2017.

[40] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *ISCA 36*, 2009.

[41] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *MICRO 52*, 2019.

[42] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO 36*, 2003.

[43] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA 30*, 2003.

[44] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, 2014.

[45] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *ISCA 44*, 2017.

[46] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA 44*, 2017.

[47] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 402–415, IEEE, 2021.

[48] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector rda for sparsity," *arXiv preprint arXiv:2104.12760*, 2021.